09 | 用户态跟踪：如何使用 eBPF 排查应用程序？

2022-02-04 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >

讲述：山荣
时长 20:32 大小 18.81M

你好，我是倪朋飞。

前面两讲，我带你梳理了查询 eBPF 跟踪点的常用方法，并以短时进程的跟踪为例，通过 bpftrace、BCC 和 libbpf 等三种方法实现了短时进程的跟踪程序。学完这些内容，我想你已经可以根据自己的实际需求，查询到内核跟踪点或内核函数，并自己开发一个 eBPF 内核跟踪程序。

也许你想问：我们能不能利用与跟踪内核状态类似的方法，去跟踪用户空间的进程呢？答案是肯定的。只要把内核态跟踪使用的 kprobe 和 tracepoint 替换成 uprobe（或者用户空间定义的静态跟踪点（User Statically Defined Tracing，简称 USDT），并找出用户进程需要跟踪的函数，作为 eBPF 程序的挂载点，你就可以去跟踪用户进程的内部状态。

那具体该怎么做呢？今天，我就带你一起来看看，如何使用 eBPF 去跟踪用户进程的执行状态。

如何查询用户进程跟踪点？

在 07 讲 中我曾提到，在跟踪内核的状态之前，你需要利用内核提供的调试信息查询内核函数、内核跟踪点以及性能事件等。类似地，在跟踪应用进程之前，你也需要知道**这个进程所对应的二进制文件中提供了哪些可用的跟踪点**。那么，从哪里可以找到这些信息呢？如果你使用 GDB 之类的应用调试过程序，这时应该已经想到了，那就是**应用程序二进制文件中的调试信息**。

在静态语言的编译过程中，通常你可以加上 -g 选项保留调试信息。这样，源代码中的函数、变量以及它们对应的代码行号等信息，就以 DWARF（Debugging With Attributed Record Formats，Linux 和类 Unix 平台最主流的调试信息格式）格式存储到了编译后的二进制文件中。

有了调试信息，你就可以通过 readelf、objdump、nm 等工具，查询可用于跟踪的函数、变量等符号列表。比如，我经常使用 readelf 命令，查询二进制文件的基本信息。在终端中执行下面的命令，就可以查询 libc 动态链接库中的符号表：

复制代码

```
# 查询符号表（RHEL8系统中请把动态库路径替换为/usr/lib64/libc.so.6）
readelf -Ws /usr/lib/x86_64-linux-gnu/libc.so.6

# 查询USDT信息（USDT信息位于ELF文件的notes段）
readelf -n /usr/lib/x86_64-linux-gnu/libc.so.6
```

当然，我们 07 讲 中提到的 bpftrace 工具也可以用来查询 uprobe 和 USDT 跟踪点，其查询格式如下所示（同样支持 * 通配符过滤）：

复制代码

```
# 查询uprobe（RHEL8系统中请把动态库路径替换为/usr/lib64/libc.so.6）
bpftrace -l 'uprobe:/usr/lib/x86_64-linux-gnu/libc.so.6:*'

# 查询USDT
bpftrace -l 'usdt:/usr/lib/x86_64-linux-gnu/libc.so.6:*'
```

同内核跟踪点类似，你也可以加上 -v 选项查询用户探针的参数格式。不过需要再次强调的是，**想要通过二进制文件查询符号表和参数定义，必须在编译的时候保留 DWARF 调试信息**。

除了符号表之外，理论上你可以把 uprobe 插桩到二进制文件的任意地址。不过这要求你对应用程序 ELF 格式的地址空间非常熟悉，并且具体的地址会随着应用的迭代更新而发生变化。所以，在需要跟踪地址的场景中，一定要记得去 ELF 二进制文件动态获取地址信息。

另外需要提醒你的是，**uprobe 是基于文件的。当文件中的某个函数被跟踪时，除非对进程 PID 进行了过滤，默认所有使用到这个文件的进程都会被插桩。**

编程语言会影响 eBPF 的跟踪吗？

如果你了解过不同的编程语言，看到这里你肯定会想到，这些查询跟踪点信息的方法很可能是跟编程语言相关的。C 语言和 Python 语言是我们课程的学习基础，也是前面几讲的案例中反复使用的编程语言，我相信你至少已经对它们有了简单的了解。这里我们对比下这两种编程语言在运行方式上的区别：

C 语言程序需要编译为二进制文件之后再执行，编译时加上 -g 选项就可以在最终的二进制文件中保留调试信息；

Python 语言程序则是一个文本文件，不需要 C 语言那样的编译过程，而是由 Python 解释器进行语法分析后执行。因而，上述readelf 等工具没法从 python 二进制文件中直接读取到应用程序的调试信息。

当然，C 和 Python 只是我们课程中使用的两种编程语言，实际的编程语言种类要多得多。如果把常用的编程语言进行归类，按照其运行原理，我认为大致上可以分为三类：

第一类是 C、C++、Golang 等编译为机器码后再执行的**编译型语言**。这类编程语言开发的程序，通常会编译成 ELF 格式的二进制文件，包含了保存在寄存器或栈中的函数参数和返回值，因而可以直接通过二进制文件中的符号进行跟踪。

第二类是 Python、Bash、Ruby 等通过解释器语法分析之后再执行的**解释型语言**。这类编程语言开发的程序，无法直接从语言运行时的二进制文件中获取应用程序的调试信息，通常需要跟踪解释器的函数，再从其参数中获取应用程序的运行细节。

最后一类是 Java、.Net、JavaScript 等先编译为字节码，再由即时编译器（JIT）编译为机器码执行的**即时编译型语言**。同解释型语言类似，这类编程语言无法直接从语言运行时的二进制文件中获取应用程序的调试信息。跟踪 JIT 编程语言开发的程序是最困难的，因为 JIT 编译的状态只存在于内存中。

对比这几类编程语言，你可以发现，编程语言的类型对 eBPF 跟踪也有非常大的影响，不同类型编程语言开发的应用程序，其跟踪过程和难度也不相同。接下来，我就用几个案例带你一起来看看每一类编程语言的跟踪方法。

跟踪编译型语言应用程序

由于可以直接从调试信息中获取到符号（用于跟踪函数）和帧指针（用于跟踪调用栈），编译型语言开发的应用程序是比较容易跟踪的。

不过需要注意的是，大部分编译型语言遵循 ABI（Application Binary Interface） 调用规范，函数的参数和返回值都存放在寄存器中。而 Go 1.17 之前使用的是 Plan 9 调用规范，函数的参数和返回值都存放在堆栈中；直到 1.17， Go 才从 Plan 9 切换到 ABI 调用规范。所以，在跟踪函数参数和返回值时，你需要**首先区分编程语言的调用规范**，然后再去寄存器或堆栈中读取函数的参数和返回值。

此外，**调试信息并非一定要内置于最终分发的应用程序二进制文件中，它们也可以放到独立的调试文件存储**。为了减少应用程序二进制文件的大小，通常会把调试信息从二进制文件中剥离出来，保存到 `<应用名>.debuginfo` 或者 `<build-id>.debug` 文件中，后续排查问题需要用到时再安装。

比如，在 RHEL 和 Ubuntu 等常见的 Linux 发行版中，调试信息跟应用程序通常是两个不同的软件包。而对很多开发者来说，每次编译和发布应用程序之前，通常都需要执行一下 `strip` 命令，把调试信息删除后才发布到生产环境中。

这里给你个小提示：ELF 符号表包含 `.symtab`（应用本地的符号）和 `.dynsym`（调用到外部的符号），`strip` 命令实际上只是删除了 `.symtab` 的内容。

接下来，我就以每个 Linux 用户都会使用的 Bash 为例（Bash 是一个典型的 C 语言程序），带你一起看看，如何跟踪编译型语言应用程序的执行状态。

在服务器的维护过程中，系统维护人员都需要审计每个用户登录后都执行了哪些命令，以便事后排查问题时参考。由于登录后所有命令的执行都发生在 Bash 中，那么，有没有可能使用 eBPF 来跟踪 Bash 里面到底执行过什么命令呢？答案是肯定的。

在跟踪 Bash 之前，首先执行下面的命令，安装它的调试信息：

复制代码

```
# Ubuntu
sudo apt install bash-dbgsym

# RHEL
sudo debuginfo-install bash
```

有了 Bash 调试信息之后，再执行下面的几步，查询 Bash 的符号表：

复制代码

```
# 第一步，查询 Build ID（用于关联调试信息）
readelf -n /usr/bin/bash | grep 'Build ID'
# 输出示例为：
#     Build ID: 7b140b33fd79d0861f831bae38a0cdfdf639d8bc

# 第二步，找到调试信息对应的文件（调试信息位于目录/usr/lib/debug/.build-id中，上一步中得
ls /usr/lib/debug/.build-id/7b/140b33fd79d0861f831bae38a0cdfdf639d8bc.debug

# 第三步，从调试信息中查询符号表
readelf -Ws /usr/lib/debug/.build-id/7b/140b33fd79d0861f831bae38a0cdfdf639d8bc
```

参考 Bash 的源代码，每条 Bash 命令在运行前，都会调用 `char*readline (const char*prompt)` 函数读取用户的输入，然后再去解析执行（Bash 自身是使用编译型语言 C 开发的，而 Bash 语言则是一种解释型语言）。

注意，`readline` 函数的参数是命令行提示符（通过环境变量 `PS1`、`PS2` 等设置），而返回值才是用户的输入。因而，我们只需要跟踪 `readline` 函数的返回值，也就是使用 `uretprobe` 跟踪。

bpftrace、BCC 以及 libbpf 等工具均支持 `uretprobe`，因而最简单的跟踪方法就是使用 bpftrace 的单行命令：

```
复制代码
1 sudo bpftrace -e 'uretprobe:/usr/bin/bash:readline { printf("User %d executed
```

这个命令中具体内容的作用如下：

`uretprobe:/usr/bin/bash:readline` 设置跟踪类型为 `uretprobe`，跟踪的二进制文件为 `/usr/bin/bash`，跟踪符号为 `readline`；

中括号里的内容为 uretprobe 的处理函数；

处理函数中，`uid` 和 `retval` 是两个内置变量，分别表示用户 UID 以及返回值；

`str` 用于从指针中读取字符串，`str(retval)` 就是 Bash 中输入命令的字符串；

`printf` 用于向终端中打印一个字符串。

打开一个终端，并在新终端中执行 `ps` 命令，然后就会在第一个终端中看到如下的输出（即 UID 为 1000 的用户执行了 `ps` 命令）：

```
复制代码
1 Attaching 1 probe...
2 User 1000 executed "ps" command
```

和上一讲的案例类似，这里我们也可以使用 BCC 和 libbpf 实现相同的跟踪程序。由于它们的实现逻辑是类似的，并且 BCC 提供了对初学者更友好的接口，所以接下来，我就以 BCC 为例，带你一起看看如何来跟踪用户空间的 Bash。

还记得用 BCC 来开发一个 eBPF 跟踪程序的具体步骤吗？如果你不记得，也没关系，我来带你简单回顾一下。BCC 的使用可以分为两部分：

第一部分是用 C 语言开发的 eBPF 程序。在 eBPF 程序中，你可以利用 BCC 提供的库函数和宏定义简化你的处理逻辑。

第二部分是用 Python 语言开发的前端界面，其中包含 eBPF 程序加载、挂载到内核函数和跟踪点，以及通过 BPF 映射获取和打印执行结果等部分。在前端程序中，你同样可以利用 BCC 库来访问 BPF 映射。

对于第一部分来说，我们要做的就是从 uretprobe 的处理函数中获取 `readline` 的返回值，然后提交到性能事件映射中。这里你可能想问：如何获取返回值呢？是不是已经有库函数可以直接拿过来用呢？对此，特别提醒你一点：**当碰到不懂的问题，特别是不清楚接口调用的具体格式时，不要去互联网搜索，而是应该查询官方文档（或者源代码）来确认。**

对于 BCC 的 uretprobe 来说，其官方文档链接就是 uretprobes。根据这里的文档，uretprobe 处理函数的定义格式应该为 `function_name(struct pt_regs *ctx)`，而返回值可以通过宏 `PT_REGS_RC(ctx)` 来获取（实际上，kretprobe 也是采用相同的格式）。有了这些文档，就可以按照这里的函数格式来完成第一部分的开发了。代码如下所示，每一步我都加了详细的注释：

复制代码

```
// 包含头文件
#include <uapi/linux/ptrace.h>

// 定义数据结构和性能事件映射
struct data_t {
    u32 uid;
    char command[64];
};
BPF_PERF_OUTPUT(events);

// 定义uretprobe处理函数
int bash_readline(struct pt_regs *ctx)
{
    // 查询uid
    struct data_t data = { };
    data.uid = bpf_get_current_uid_gid();

    // 从PT_REGS_RC(ctx)读取返回值
    bpf_probe_read_user(&data.command, sizeof(data.command), (void *)PT_REGS_R

    // 提交性能事件
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

从代码中你可以看到，eBPF 程序的结构跟上一讲基本类似，唯一需要注意的是函数的定义格式以及返回值的读取位置。将上述代码保存到 `bashreadline.c` 文件中，我们就完成了第一部分的开发。

有了 eBPF 程序之后，第二部分的 Python 前端也比较直观，代码如下所示：

复制代码

```
# 引入BCC库
from bcc import BPF
from time import strftime

# 加载eBPF 程序
b = BPF(src_file="bashreadline.c")

# 挂载uretprobe
b.attach_uretprobe(name="/usr/bin/bash", sym="readline", fn_name="bash_readlin

# 定义性能事件回调（输出时间、UID以及Bash中执行的命令）
def print_event(cpu, data, size):
    event = b["events"].event(data)
    print("%-9s %-6d %s" % (strftime("%H:%M:%S"), event.uid, event.command.dec

# 打印头
print("%-9s %-6s %s" % ("TIME", "UID", "COMMAND"))

# 绑定性能事件映射和输出函数，并从映射中循环读取数据
b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

可以看到，这部分的代码跟上一讲案例的结构也是类似的。主要的不同点在于，挂载 uretprobe 的时候调用了 `b.attach_uretprobe()` ，并在其参数中传入了二进制文件的路径 `name="/usr/bin/bash"` ，以及要跟踪的符号 `sym="readline"`。

在 BCC 的内部，当你挂载 uprobe 或者 uretprobe 到用户程序时，由于同一个符号可能出现多次，BCC 会首先查询该符号的所有地址，然后把 eBPF 程序挂载到所有不同的地址上。

如果你没有使用 BCC 等库，你就需要自己来实现类似的逻辑。而要实现它，还需要详细了解 ELF 文件格式，以及对应库函数的使用方法，这个难度就要大很多了。这也是我推荐你使用 BCC、libbpf 等库进行 eBPF 程序开发的原因。这些库已经帮你把很多繁杂且跟 eBPF 事件处理主要逻辑无关的部分实现好了，你只需要调用它们实现你最核心的 eBPF 处理逻辑即可。

将上述代码保存到 bashreadline.py ，然后执行 sudo python3 bashreadline.py ，我们就可以运行这个跟踪程序。打开一个新终端并运行 ls 命令，回到第一个终端，你就可以看到如下的输出：

复制代码

```
TIME      UID    COMMAND
07:17:07  1000   ls
```

从这里的例子中，你可以发现：编译型语言应用程序的跟踪与内核的跟踪是类似的，只不过是把跟踪类型从 kprobe 换成了 uprobe 或者 USDT（USDT 的例子我会在接下来的内容中讲到）。不同的地方在于符号信息：应用程序的符号信息可以存放在 ELF 二进制文件中，也可以以单独文件的形式，放到调试文件中；而内核的符号信息除了可以存放到内核二进制文件中之外，还会以 /proc/kallsyms 和 /sys/kernel/debug 等形式暴露到用户空间。

跟踪解释型语言应用程序

了解了编译型语言应用程序的跟踪方法之后，接下来，我们再来看看解释型语言应用程序又该如何跟踪。

上面我已经提到，你无法从解释型语言的二进制文件中直接获取应用程序的调试信息，而只能获得解释器本身的符号信息。所以，对于这类语言开发的应用程序，通常需要跟踪解释器内的函数，再从其参数中获取应用程序的运行细节。

对于各种解释型编程语言的二进制文件（如 Python、PHP 等），你可以使用类似编译型语言应用程序的跟踪点查询方法，查询它们在解释器层面的 uprobe 和 USDT 跟踪点。比如，对于 Python3 来说，你可以执行下面的命令查询：

复制代码

```
sudo bpftrace -l '*:/usr/bin/python3:*'
```

命令执行后，你会得到 1500 多个跟踪点。接下来的难点在于，**如何从这些解释器的跟踪点中找出应用程序的函数信息**，所以你需要对解释器的运行原理有一定的了解。

既然我们这门课中的案例大量使用了 Python 这种解释型编程语言，那我们下面就来试试，能不能从 Python 解释器的跟踪点中，跟踪应用程序的函数执行过程。

实际上，根据 Python 文档，为其开启 USDT 跟踪点（编译选项为 `--with-dtrace`）之后，Python3 二进制文件中就会包含一系列的 USDT 跟踪点。这些跟踪点也可以通过 bpftrace 查询到：

复制代码

```
$ bpftrace -l '*:/usr/bin/python3:*'
usdt:/usr/bin/python3:python:audit
usdt:/usr/bin/python3:python:function__entry
usdt:/usr/bin/python3:python:function__return
usdt:/usr/bin/python3:python:gc__done
usdt:/usr/bin/python3:python:gc__start
usdt:/usr/bin/python3:python:import__find__load__done
usdt:/usr/bin/python3:python:import__find__load__start
usdt:/usr/bin/python3:python:line
```

其中，跟函数调用相关的正是 `function__entry` 和 `function__return`，因而它们就可以用来跟踪函数的调用过程。根据 Python 文档，这两个函数的定义格式为：

复制代码

```
// 三个参数分别是文件名、函数名和行号
function__entry(str filename, str funcname, int lineno)
function__return(str filename, str funcname, int lineno)
```

有了跟踪点的定义格式，我们就可以使用 eBPF 来跟踪这些函数。比如，对 `function__entry` 来说，执行下面的 bpftrace 单行命令，就可以跟踪 Python 函数的调用信息：

复制代码

```
sudo bpftrace -e 'usdt:/usr/bin/python3:function__entry { printf("%s:%d %s\n",
```

在这个命令中，`arg0`、`arg1`、`arg2` 表示函数的三个参数。这条命令的含义就是把这几个参数拼接成 `文件名:行号 函数名` 的格式，然后再打印到终端上。

打开一个新终端，并执行下面的 Python 命令开启一个 http 服务：

```
python3 -m http.server 8080
```

然后，再回到第一个终端，就可以看到如下的输出：

```
...
/usr/lib/python3.9/socketserver.py:254 service_actions
/usr/lib/python3.9/selectors.py:403 select
/usr/lib/python3.9/socketserver.py:254 service_actions
/usr/lib/python3.9/selectors.py:403 select
```

恭喜，到这里，你已经成功从 Python 解释器中跟踪到了应用程序中的函数调用。

对于其他的解释型编程语言，其跟踪过程也是类似的，只是要把跟踪函数换成该语言解释器中处理函数调用的跟踪点。如果你不了解它们底层的实现原理，可以参考 BCC 的 ucalls 和 uflow ，开发针对它们的跟踪程序。

刚才我们使用的是 bpftrace，如果你想使用 BCC 和 libbpf 来开发更复杂的跟踪功能也是没问题的。比如，还是以 BCC 为例，它的跟踪过程与编译型语言相比，主要有两个不同点，下面我们来具体看看。

第一，在 eBPF 程序部分，USDT 跟踪点需要调用 `bpf_usdt_readarg()` 函数，来读取函数参数（对于指针类数据，还需调用 `bpf_probe_read_user()` 读取指针指向的内容），代码如下所示：

```
// 头文件引用和数据结构定义...

int print_functions(struct pt_regs *ctx)
{
    uint64_t argptr;
    struct data_t data = { };

  // 参数1是文件名
```

```
    bpf_usdt_readarg(1, ctx, &argptr);
    bpf_probe_read_user(&data.filename, sizeof(data.filename),
                (void *)argptr);

  // 参数2是函数名
    bpf_usdt_readarg(2, ctx, &argptr);
    bpf_probe_read_user(&data.funcname, sizeof(data.funcname),
                (void *)argptr);

  // 参数3是行号
  bpf_usdt_readarg(3, ctx, &data.lineno);

    // 最后提交性能事件
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
};
```

第二，在 Python 前端程序中，挂载跟踪点时需要换成 USDT 跟踪点，即：

复制代码

```
from bcc import BPF, USDT

u = USDT(pid=pid)
u.enable_probe(probe="function__entry", fn_name="print_functions")
b = BPF(src_file="<ebpf-program>.c", usdt_contexts=[u])

# 其他处理逻辑
```

除了这两个不同点，其他的逻辑都是类似的。那么，参考上述的编译型语言跟踪程序，你能在这两个不同点的基础上，写出完整的 BCC 跟踪程序吗？这里你可以先想一想，这也是今天留给你的思考题。欢迎你在课后跟我分享你的思路和实现方法。

跟踪即时编译型语言应用程序

除了上面两类编程语言，还有一种是 Java、.Net 等即时编译型语言。对于这类编程语言开发的应用程序，应用源代码会先编译为字节码，再由即时编译器（JIT）编译为机器码执行。以 Java 为例，Java 虚拟机（JVM）除了会执行常规的 JIT 即时编译之外，还会在执行过程中对运行流程进行剖析和优化，因而也加大了跟踪的难度。

同解释型编程语言类似，uprobe 和 USDT 跟踪只能用在即时编译器上，从即时编译器的跟踪点参数里面获取最终应用程序的函数信息。由于 USDT 跟踪点比 uprobe 更为稳定，

如果编程语言提供了 USDT 跟踪功能，我推荐打开 USDT 跟踪（比如 Java 需要打开 `--enable-dtrace` 编译选项），再利用 USDT 而不是 uprobe 去跟踪应用的执行过程。

要找出即时编译器的跟踪点同应用程序运行之间的关系，就需要你对编程语言的底层运行原理非常熟悉，这也是跟踪即时编译型语言应用程序最难的一步。不过这一步梳理清楚之后，具体的跟踪步骤与解释型编程语言应用程序是类似的。

如果你需要跟踪这类编程语言开发的应用，可以参考 BCC 提供的一系列用户态跟踪库，这里面已经包含了常见的几种编程语言的适配，具体的跟踪步骤我在这里就不展开了。

小结

今天，我带你一起梳理了应用进程的跟踪方法，并以 Bash 和 Python 这两种编程语言为例，带你开发了它们的 uprobe 和 USDT 等不同类型的跟踪程序。

在跟踪应用进程之前，你需要先获取这个进程所对应的二进制文件的跟踪点，以及这些跟踪点同应用程序函数的对应关系。**这个对应关系依赖于编程语言的类型**：编译型语言开发的程序中直接包含了应用程序的符号信息，因而可以直接拿来跟踪；而解释型语言和即时编译型语言的二进制文件，只包含了解释器或即时编译器的符号信息，所以应用程序的运行状态还需要从解释器或即时编译器跟踪的参数中去获取。

在这一讲结束之前，我还想提醒你：**用户进程的跟踪，本质上是通过断点去执行 uprobe 处理程序。**虽然内核社区已经对 BPF 做了很多的性能调优，跟踪用户态函数（特别是锁争用、内存分配之类的高频函数）还是有可能带来很大的性能开销。因此，我们在使用 uprobe 时，应该尽量避免跟踪高频函数。

思考题

在跟踪 Python 语言程序的 BCC 案例中，我列出了它在使用 USDT 跟踪时，相对于编译型语言的两个不同点：

eBPF 程序中需要调用 `bpf_usdt_readarg()` 来读取参数；

Python 前端中需要挂载 USDT 跟踪点。

根据这些提示，以及上面的编译型语言 BCC 跟踪程序，你能试着写出完整的 BCC 跟踪程序吗？欢迎在评论区和我分享你的思路和解决方法。

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 内核跟踪（下）：开发内核跟踪程序的进阶方法

下一篇 10 | 网络跟踪：如何使用 eBPF 排查网络问题？

精选留言 (3)

写留言

莫名

2022-02-04

1. 跟踪编译型语言应用程序以 Bash Shell 作为例子，觉得有些容易与解释型语言混淆。解释型语言的特征是解释程序自身为编译型程序，利用自身内置的函数进行语法分析和执行，Bash readline 函数的功能即是如此。以编译型语言 Golang helloworld 程序使用 uprobes 或者开源的 Nginx 使用 USDT 可能更恰当一些。

...

展开

作者回复: 1. 其实最终还是要看跟踪的目的是什么：如果是要排查Bash内部的执行过程，那就是跟踪编译型语言；但要是排查SHELL脚本的执行过程，那就变成了跟踪解释性语言。

2. 非常棒的答案！

3

Geek_59a6f9

2022-02-04

用户态进程跟踪是不是使用frida效果更好？

作者回复: 嗯嗯，展示效果都是可以在用户态程序中去自定义的。

1

ZR2021

2022-02-04

老师，跟踪用户态程序的时候还有512字节的限制吗

作者回复: eBPF 栈空间的限制跟要跟跟踪的事件是没关系的，但其实都可以通过映射避免内存限制的问题。

共 2 条评论 >

1